VŠB — Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Automatické strukturování kódu programu

Automatic Structuring of Program Code

2012/2013 Hamid Khairy

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Hamid Khairy**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Automatické strukturování kódu programu
Automatic Structuring of Program Code

Zásady pro vypracování:

Cílem práce je implementovat algoritmus, popsaný ve článku Baker (1977), který pro zadaný nestrukturovaný kód programu, ve kterém je veškeré řízení realizováno pomocí goto a podmíněných skoků, vytvoří kód využívající strukturované konstrukce (bloky, if then else, while, atd.) takovým způsobem, aby výsledný kód přehledně zachycoval skutečnou strukturu řídícího toku programu a aby byl minimalizován počet použitých příkazů goto.

1. Nastudujte algoritmus popsaný v Baker (1977).
2. Algoritmus implementujte.
3. Implementaci otestujte na vhodně zvolených testovacích datech.

Seznam doporučené odborné literatury:

B. S. Baker: An Algorithm for Structuring Flowgraphs, Journal of the Association for Computing Machinery (JACM), Volume 24, Issue 1, January 1977, pp. 98-120.

Další literatura podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Zdeněk Sawa, Ph.D.**

Datum zadání: 18.11.2011

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Datum: 5. 5. 2013

Rád bych poděkoval vedoucímu bakalářské práce Ing. Zdeňku Sawovi, Ph.D. za pomoc porozumění problému této práce a rady při její tvorbě.

Abstrakt

Cílem práce je implementovat algoritmus, popsaný ve článku Baker(1977), který pro zadaný nestrukturovaný kód programu, ve kterém je veškeré řízení realizováno pomocí goto a podmíněných skoků, vytvoří kód využívající strukturované konstrukce (bloky, if then else, while, atd.) takovým způsobem, aby výsledný kód přehledně zachycoval skutečnou strukturu řídícího toku programu a aby byl minimalizován počet použitých příkazů goto.

Klíčová slova: c++, graf toku, redukovatelnost grafu, follow, dominance grafu

Abstract

This bachelor's thesis aim is to implement algorithm, described in article Baker(1977), which for entered non-structured program code in which all control is realised by goto and conditional jump statements, creates code using structured constructs (blocks, if then else, while, etc.) in such a way that resulting code will clearly transcribe real structure of program control flow and minimalize ammount of used goto statements.

Klíčová slova: c++, flow graph, graph reducibility, follow, graph dominance

Obsah

1 Úvod

V dnešní době existují mnohé způsoby a paradigmata, jež se v programování používají. Jedním ze starších, ale stále často používaným paradigmatem je i strukturované programování. Tento způsob programování se vyznačuje velkým využitím blokových příkazů se snahou o co nejmenší využití instrukcí skoku.

Nicméně ne všechny jazyky takovéto konstrukce obsahují. Kód napsaný s extenzivním využitím instrukcí skoku bývá velice nečitelný a velmi časově nákladný na údržbu.

Cílem této práce je vytvořit program implementující algoritmus popsaný v článku [1], který pro „nestrukturovaný kód“ vytvoří odpovídající strukturovaný kód s využitím blokových instrukcí a co nejmenším počtem instrukcí skoku .

2 Specifikace problému

2.1 Rozbor problému

V dobách prvních generací počítačů bylo paradigma nestrukturovaného programování bráno jako samozřejmost a jazyky v té době nic jiného ani neumožňovaly. S postupným nárustem výpočetního výkonu se však stále častěji začalo narážet na časté problémy a nedostatky tohoto paradigmatu, mezi které patří špatná čitelnost kódu a jeho potencionální chybovost.

Strukturované programování v těchto aspektech představilo revoluci, kdy byly výsledné programy nejen značně přehlednější, ale především jejich chybovost byla oproti nestrukturované verzi zanedbatelná.

2.2 Možné způsoby využití práce

Potřeba převést kód na jeho strukturovanou variantu může vyvstat z několika důvodů. V první řadě se může jednat o program, jež byl napsán v jednom ze starších jazyků, které neumožňují nebo alespoň ve své době neumožňovaly použití blokových instrukcí. (assembler, COBOL, prvotní verze Fortranu, prvotní verze Basicu atp.)

Další z možných důvodů využití této práce může být její užití jako jedna z posledních fází běhu dekompilátoru.

3 Specifikace zdrojového jazyka

Jedním z možných vstupů programu je vložení zdrojového kódu, ze kterého se poté sestaví potřebný vstupní graf. Proto bylo třeba vytvořit odpovídající jazyk, který půjde snadno převést na graf dle daných požadavků.

3.1 Základní požadavky jazyka

Jazyk je mírně upravená verze jazyka, popsaného v článku jako SL. Jeho základním požadavkem je aby neobsahoval větvící ani blokové instrukce což umožní snadný převod kódu do grafu toku.

3.2 Typy instrukcí jazyka

- **Typ Start:** Instrukce start značí počátek programu a potažmo i první uzel v grafu. Do instrukce start nevedou žádné skoky a vede z ní pouze jediná cesta. Start není nutné zadávat, pokud v programu není, bude do něj doplněna automaticky. Pokud je instrukce start zadána, musí být na počátku programu.
- **Typ Stop:** Podobná instrukci start je instrukce stop. Stop označuje ukončení programu a také konečný uzel grafu. Z instrukce stop již nevede žádná cesta do jiného uzlu grafu.
- **Typ SLC:** Instrukce SLC představuje většinu kódu programu. SLC představuje jakýkoliv výraz, jež se sekvenčně vykoná a poté pokračuje následující instrukcí v pořadí. To může být výrok, ale i třeba volání funkce (pro jednoduchost je možno do vstupního kódu programu vkládat pouze funkce bez parametrů). Může do nich v grafu vést vicero hran, z nich samotných však vede pouze jedna.
- **Typ Goto:** Instrukce goto je instrukci skoku. Je podobná instrukci SLC s tím rozdílem, že nemusí vést na následující instrukci v pořadí. Label, určující cíl skoku musí být v programu zadán. Program na vstupu v sobě musí obsahovat instrukci s labelem odpovídající použitému labelu v instrukci goto.
- **Typ Conditional:** Instrukce conditional je instrukci podmíněného skoku. Pokud je obsažená podmínka splněna, proběhne skok na instrukci určenou labelem v opačném případě pokračuje program sekvenčně dál. Program na vstupu v sobě musí obsahovat instrukci s labelem odpovídající použitému labelu v instrukci conditional.
- **Typ Label:** Label není samostatnou instrukcí, ale jedná se určení „názvu“ instrukce. Dle názvu se poté určuje cílový skok instrukcí goto a conditional. Jedna instrukce může mít více labelů.

4 Překlad zdrojového jazyka

Následující odstavce popisují způsob překladu zdrojového jazyka na vstupní graf toku.

4.1 Lexikální analýza

Prvním z kroků tvorby grafu je lexikální analýza, kterou se rozumí převod kódu na sekvenci symbolů (tokenů). Ta čte ze vstupu po znacích a v případě, že nalezne znak či slovo odpovídající jednomu ze symbolů (viz. Tabulka 4.1), tak jej přidá do pole symbolů.

Symbol	Popis
TOKEN_LESSTHAN	Operátor <
TOKEN_LESSTHANEQUAL	Operátor <=
TOKEN_GREATERTHAN	Operátor >
TOKEN_GREATERTHANEQUAL	Operátor >=
TOKEN_EQUAL	Operátor ==
TOKEN_NOT	Operátor !
TOKEN_LEFT_BRACKET	Znak (
TOKEN_RIGHT_BRACKET	Znak)
TOKEN_COLON	Znak :
TOKEN_IF	Klíčové slovo if
TOKEN_THEN	Klíčové slovo then
TOKEN_ELSE	Klíčové slovo else
TOKEN_START	Klíčové slovo start
TOKEN_STOP	Klíčové slovo stop
TOKEN_GOTO	Klíčové slovo goto
TOKEN_IDENTIFIER	Identifikátor
TOKEN_NUMBER	Celočíselná hodnota
TOKEN_ASSIGNMENT	Operátor =
TOKEN_DELIMITER	Oddělovač ;
TOKEN_NOTEQUAL	Operátor !=
TOKEN_PLUS	Operátor +
TOKEN_MINUS	Operátor -
TOKEN_MULTIPLY	Operátor *
TOKEN_DIVIDE	Operátor /

Tabulka 4.1: Tabulka symbolů

4.2 Syntaktická analýza

Jakmile máme k dispozici sekvenci symbolů z předchozí lexikální analýzy můžeme začít provádět analýzu syntaktickou a s ní v tomto případě spojený i převod symbolů do grafu. Syntaktická analýza má za úkol ověřit jestli zadaný vstupní kód je napsán správně.

4.2.1 Gramatika vstupního jazyka v Bacus-Naurově formě

```
<program>         ::= <start>|<label><instruction>
<start>           ::= „start;“
<label>           ::= „“ | <identifier> “:“
<instruction>     ::= <stop>|<SLC>|<conditional>|<goto>
<stop>            ::= „stop;“
<SLC>             ::= <expression> | <func_call>
<conditional>     ::= “if (” <expression>|<func_call> “)” <goto>
<goto>            ::= „goto „ <identifier> „;“
<expression>      ::= <operator_not><expression>
                    | <statement> <operator> <statement>
<statement>       ::= <identifier> | <number>
<func_call>       ::= <identifier> “()”;
```

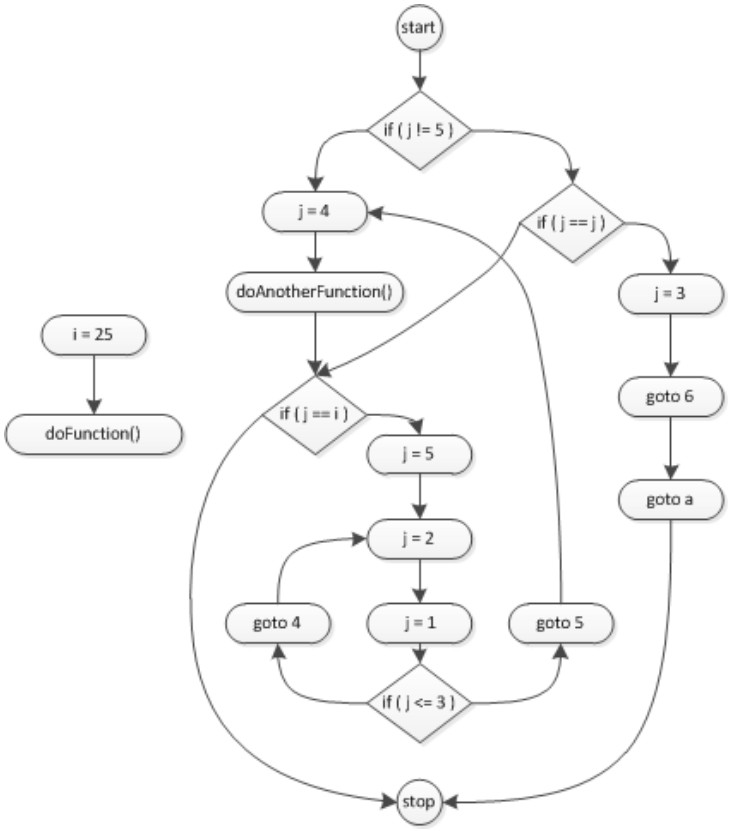
Příklad platného vstupu 4.1:

```
if (j!=5) goto 5;
if (j==j) goto 11;
j=3;
goto 6;
i = 25;
doFunction();
6:goto a;
5:j=4;
doAnotherFunction();
11:if (j==i) goto a;
j=5;
15:
20:
4:j=2;
j = 1;
if ( j <= 3 ) goto 4;
goto 5;
a:s
```

4.2.2 Překlad do grafu

Jelikož algoritmus popsaný v článku [1] pracuje s grafem toku je nutné kód přeložit na onen graf. Pro tyto účely je v této práci jeho tvorba spojená se syntaktickou analýzou. Každý příkaz, vyhodnocený jako správný musí odpovídat jednomu uzlu ve vytvořeném grafu.

Proces syntaktické analýzy prochází sekvenci symbolů lineárně s využitím znalosti následujícího symbolu. V případě, že nalezne instrukci label, poznačí si že následně vytvořený uzel má alespoň jednu instrukci label a přidá ji do pole … . V momentě kdy syntaktická analýza plně identifikuje jeden z typů instrukcí, přidá do výsledného grafu uzel s typem dle instrukce a naplní jej patřičnými údaji. Po vytvoření uzlů v grafu jej musíme projít podruhé a nastavit každému uzlu správnou cestu.



Obrázek 4.1: Graf toku vytvořený z kódu v příkladu 4.1

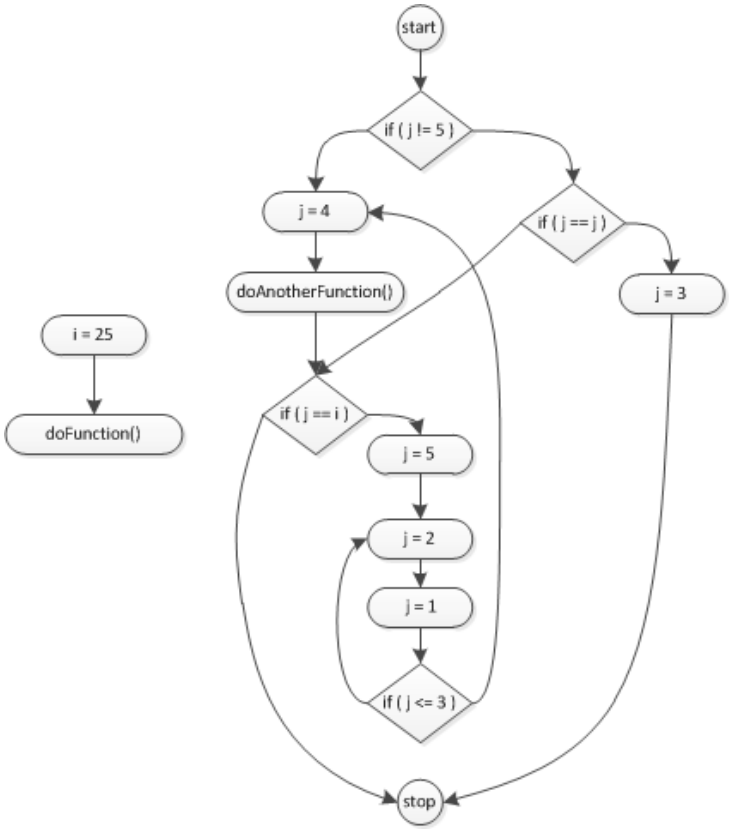
5 Zjednodušení vstupního grafu

Než přistoupíme k samotnému provedení algoritmů z článku [1] je nasnadě provést pár úprav v grafu, abychom snížili dobu provádění algoritmu.

5.1 Odstranění uzlů skoku

Ve vstupním grafu může být mnoho uzlů typu goto. Pro následné zpracování grafu je však zbytečné tyto uzly v grafu ponechat a riskovat u některých větších vstupů zpomalení algoritmu.

Pokud chceme odstranit uzly skoku musíme zajistit, aby všechny uzly za nimiž následuje skok vedly na cíl daného skoku. Toto nám však poněkud zkomplikuje případ, kdy jeden skok vede na jiný. Proto budeme postupovat pro odstranění uzlů skoku ve dvou krocích.



Obrázek 5.1: Předešlý graf s odstraněnými uzly skoku

V prvním kroku pomocí průchodu do hloubky najdeme všechny uzly skoku a určíme jim jako cíl první neskokový uzel v cestě. Poté průchodem zpět nastavíme všem uzlům skoku cíl skoku, na který ukazují. Tímto způsobem máme zaručeno, že všechny uzly skoku budou ukazovat na první neskokový uzel v cestě.

V druhém kroku lineárně projdeme graf, nastavíme správnou cestu veškerým uzlům, které vedou na uzel skoku a odstraníme uzly skoku.

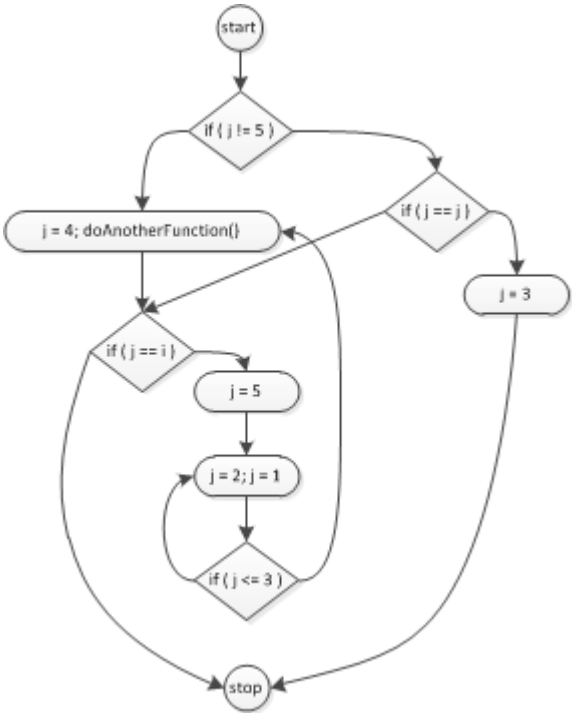
5.2 Sloučení uzlů SLC a odstranění nedosažitelných uzlů

Odstraněním uzlů skoku jsme zredukovali graf, nicméně je zde stále prostor pro hlubší redukci. Můžeme předpokládat, že u většiny kódů programů bude většina uzlů, odpovídajícího grafu, typu SLC. Tyto uzly, pokud jsou seřazeny sekvenčně za sebou a vede do nich pouze jedna hrana, lze sloučit do jediného uzlu. Tím zajistíme, že algoritmus z článku [1] se nebude zabývat mnoha přebytečnými uzly.

Druhým cílem tohoto kroku, je odstranění nedosažitelných uzlů. Nedosažitelné uzly nemá význam nikterak zpracovávat, jelikož k jejich vykonání nemůže v programu nikdy dojít. Uzel považujeme za dosažitelný, vede-li k němu alespoň jedna cesta z počátku grafu.

V prvním kroku potřebujeme určit, které SLC uzly můžeme odstranit a které ne. Začneme tedy procházet graf do hloubky a každý uzel označíme barvou dle jeho stavu zpracování. Bílé uzly jsou dosud nenavštívené uzly, šedé uzly jsme už navštívili, ale nezpracovali jsme všechny jejich potomky a černé uzly jsou takové, které jsme plně zpracovali. Pro určení možnosti odstranit SLC uzel budeme postupovat odzadu. (tzn. Budeme zpracovávat šedé uzly) Při zpracovávání šedých uzlů ověříme zda uzel je typu SLC, v případě že ano, ověříme jestli vede na další SLC uzel, do kterého vede pouze jedna hrana. Pokud jsou tyto podmínky splněny, přidáme veškerý kód z následníka uzlu jeho předchůdci, označíme následníka jako smazatelný uzel a nastavíme předchůdci cíl na uzel, na který ukazuje následník.

Z předchozího kroku máme zaručeno, že všechny dosažitelné uzly jsou obarveny černě. Pokud tedy se v grafu vyskytuje nějaký uzel, který je obarven bíle jedná se nedosažitelný vrchol. Lineárně tedy projdeme uzly v grafu a odstraníme bíle uzly a označené uzly. Postup barvení vrcholů využijeme ještě i v následujících krocích, takže musíme všem uzlům po dokončení tohoto kroku nastavit barvu na bílou.



Obrázek 5.2: Předešlý graf po sloučení uzlů SLC a odstranění nedosažitelných uzlů

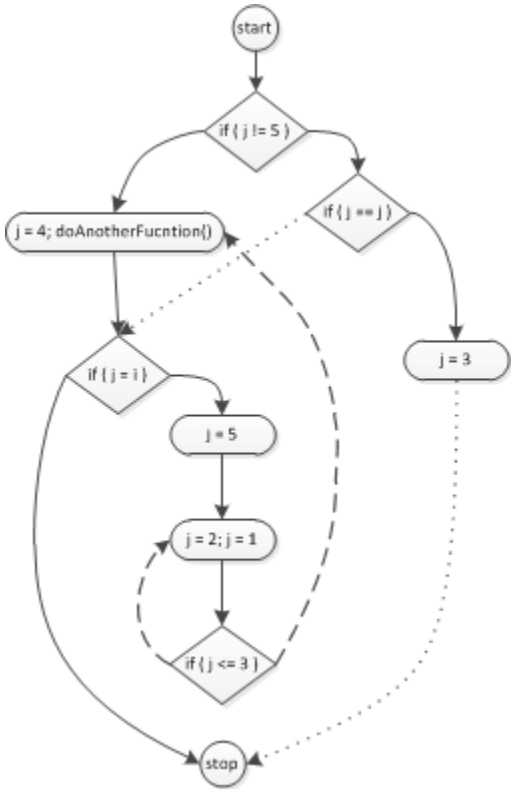
6 Uzly Repeat

V této kapitole se budeme věnovat doplnění grafu o uzly typu repeat a výpočtu důležitých vlastností těchto uzlů.

6.1 Vytvoření repeat uzlů

Abychom mohli určit kde máme vytvořit uzly repeat, potřebujeme si nějak tyto uzly definovat a určit jejich vlastnosti. Než se ale dostaneme k vlastnostem, musíme si definovat typy hran, které graf může obsahovat, jsou to:

- **Dopředná hrana**: Dopřednou hranu můžeme definovat jako hranu, která je zahrnutá v kostře grafu. Tyto hrany jsou zobrazeny v obrázku 6.1 plnou čarou.
- **Zpětná hrana**: V článku [1] je zpětná hrana definována jako: „Zpětná hrana je hrana z uzlu do sebe samotného, nebo z jeho potomka v kostře grafu.“ Tyto hrany jsou zobrazeny v obrázku 6.1 čárkovanou čarou.
- **Křížená hrana**: Křížené hrany jsou všechny hrany, které nejsou zahrnuty v kostře grafu, ale nejsou zpětné hrany. Tyto hrany jsou zobrazeny v obrázku 6.1 tečkovanou čarou.



Obrázek 6.1: Znázornění typů hran na předešlém grafu

Pro tvorbu uzlů repeat jsou pro nás momentálně klíčové zpětné hrany. Definujme si uzel p a jeho následníka q v kostře grafu. Potom platí, že pokud existuje hrana z q do p, bude p prvním uzlem cyklu.

6.1.1 Post-Order číslování

Abychom mohli určit zpětné hrany, potřebujeme zkonstruovat kostru grafu. K tomu využijeme prohledávání do hloubky s využitím zásobníku a barvením jednotlivých uzlů, podobně jako v podkapitole 5.2. Spolu s číslováním určíme uzlům i jejich rozsah. Rozsah nám určuje u každého uzlu jeho podstrom v kostře grafu.

Zprvu nastavíme hodnotu n na celkový počet uzlů v grafu – 1. Při post-order číslování budeme jednotlivé uzly v grafu číslovat odzadu. Začneme s průchodem na počátku grafu (uzel start). Ten označíme šedou barvou, přiřadíme mu rozsah, který je současná hodnota n a přidáme jej do zásobníku poté pokračujeme jeho následníkem. V případě uzlů podmínek je první vyhodnocena vždy pravdivá klauzule s jejími potomky a teprve poté je vyhodnocena nepravdivá klauzule. Takto procházíme graf až do chvíle, kdy narazíme na uzel stop. U něj rovnou zaznačíme barvu černou a nastavíme jeho hodnotu na n a dekrementujeme n o 1. Poté se vracíme zpět po uzlech uložených v zásobníku s tím, že pokud má daný uzel další následníky, začneme nejdříve zpracovávat je. Pokud nemá další nezpracované následníky, označíme jej černou barvou a nastavíme mu hodnotu na n - (1 + m), kde m je počet již nastavených uzlů. Pokud narazíme na uzel jehož následník je označen šedě, jedná se o zpětnou hranu. V takovém případě poznačíme u následníka, že se jedná o počátek cyklu a pokračujeme ve zpracovávání vlastního uzlu.

6.1.2 Vytvoření repeat uzlů

Nyní máme z předchozího kroku očíslovány všechny uzly grafu a označeny počátky cyklů, takže již můžeme přistoupit k samotnému vytvoření repeat uzlů. Než je vytvoříme, je však nutné uzly v grafu setřídit podle jim přiřazeného čísla do dočasného pole.

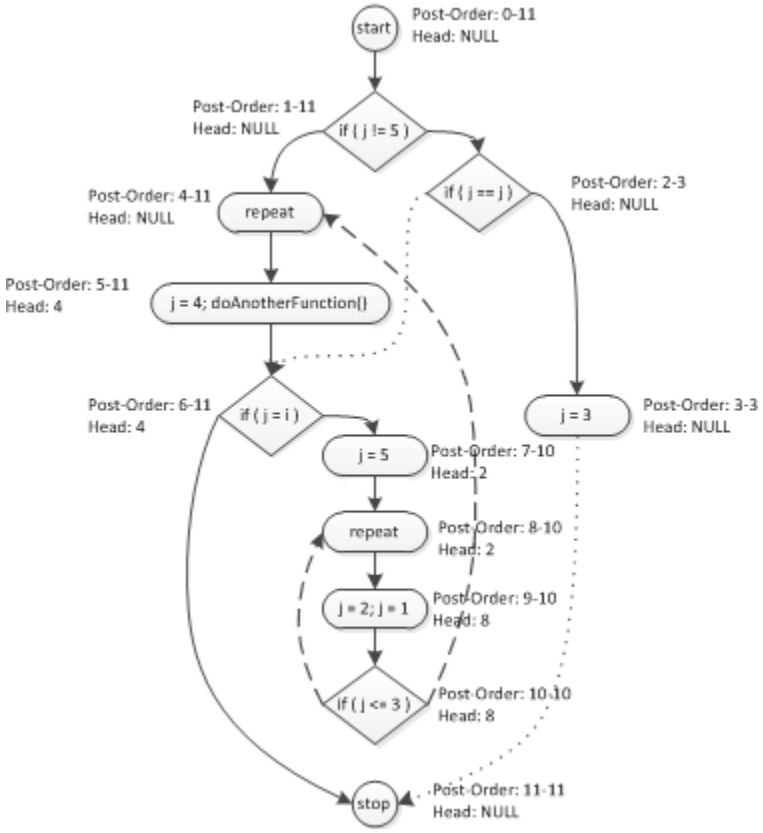
Na počátku tvorby nastavíme počet již vytvořených uzlů n na 0 a připravíme si pole uzlů o velikosti počtu uzlů v grafu + počtu uzlů označených jako počátky cyklu. Pro samotnou tvorbu začneme procházet uzly v dočasném poli lineárně. Při zpracovávání uzlu, jež nemá nastaven počátek cyklu, jej uložíme na pozici jeho čísla + n a upravíme jeho rozsah o n. Tím zachováme správné číslování i pro uzly, zpracovávané po vytvoření uzlů repeat. Když narazíme na uzel, který má nastavený počátek cyklu, vytvoříme jeho kopii, kterou nastavíme jako jeho následníka. Poté danému uzlu změníme typ na repeat, uložíme jej na pozici jeho čísla + n a upravíme rozsah o n. Následně inkrementujeme n o 1 a uložíme kopii uzlu na pozici jeho čísla + n.

Tvorbou kopie místo tvorby čistě repeat uzlu si usnadníme práci s přenastavením cílů hran, které do tohoto uzlu vedou.

6.2 Určení Head

V této fázi algoritmu máme již vytvořené uzly repeat i očíslovaný graf, nicméně stále potřebujeme jasně určit, které uzly jsou součástí těla smyčky a které už ne. Může nastat i situace, kdy je několik cyklů zanořených v sobě. V takovém případě potřebujeme navíc odlišit, kterému z cyklů daný uzel patří. K tomu nám slouží parametr head.

K výpočtu parametru head nám pomůže setřízený graf pomocí post-order číslování a znalost rozsahu daného uzlu. Zpočátku je parametr head u všech uzlů v grafu nastaven na NULL. Pro určení head začneme procházet graf lineárně. Když narazíme na uzel repeat, tak všechny hrany, které do něj vedou a jejich post-order je v rozmezí post-orderu repeat uzlu a jeho limitu (tzn. Hrana vede na uzel v rámci podstromu uzlu repeat v kostře grafu), uložíme uzel do předem připravené fronty. Poté pro každý uzel ve frontě nastavíme head na uzel repeat a projdeme všechny hrany, které do uzlu vedou a pokud bude jejich post-order v rozmezí post-orderu a jeho limitu repeat uzlu, tak je přidáme na konec fronty.



Obrázek 6.2: Předešlý graf doplněný o uzly repeat a vypočítané parametry

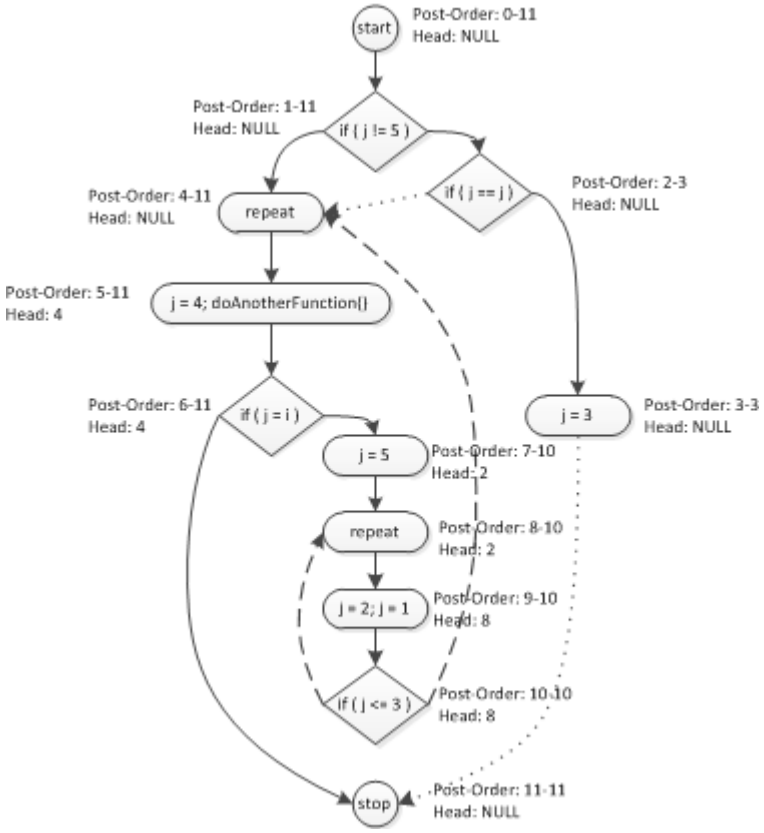
7 Množina follow

V následujících odstavcích je popsán postup pro výpočet množiny follow. Ta je pro nás klíčová při rozhodování o finálním uspořádání struktury kódu.

7.1 Tvorba redukovatelného grafu

Pro správný průběh výpočtu dominance a množiny follow je zapotřebí, aby se tyto algoritmy prováděli na redukovatelném grafu. Většina programů, které očekáváme na vstupu jsou redukovatelné, nicméně musíme počítat i s případem kdy na vstup bude zadán graf neredukovatelný a v takovém případě, kdy daný graf není redukovatelný musíme vytvořit jeho redukovatelnou alternativu, nad kterou se provedou výpočty dominance a množiny follow. Pro případ této práce můžeme předpokládat, že program je neredukovatelný v případě, kdy alespoň jedna křížená hrana vede do těla smyčky.

Prvním krokem tvorby je vytvoření identické kopie původního grafu, kterou budeme dále zpracovávat. Dále pro každý uzel v grafu ověříme, jestli jeho následník má stejný parametr head. (tzn. Jestli jeho následník je v těle stejného cyklu) Pokud nemá, vezmeme jeho parametr head a



Obrázek 7.1: Redukovatelná verze předešlého grafu

ověříme zda-li on nemá stejný parametr head. Pokud má, nastavíme jej jako nový cíl uzlu v

opačném případě pokračujeme tímto způsobem až do chvíle, kdy je tato podmínka splněna a nebo než je parametr head NULL.

7.2 Dominance v grafu

Abychom mohli vypočíst množinu follow, potřebujeme nejdříve určit dominance uzlů. Z obecné definice víme, že uzel p dominuje nad uzlem q jen a pouze v případě, kdy všechny cesty do uzlu q vedou přes uzel p. Z čehož mimo jiné vyplývá, že jeden uzel může mít vícero dominujících uzlů (tj. Dominátorů). Jednotlivé množiny dominátorů jednotlivých uzlů přehledně zachycuje tabulka 7.1.

Uzel p	DOM(p)
0	0
1	0, 1
2	0, 1, 2
3	0, 1, 2, 3
4	0, 1, 4
5	0, 1, 4, 5
6	0, 1, 4, 5, 6
7	0, 1, 4, 5, 6, 7
8	0, 1, 4, 5, 6, 7, 8
9	0, 1, 4, 5, 6, 7, 8, 9
10	0, 1, 4, 5, 6, 7, 8, 9, 10
11	0, 1, 11

Tabulka 7.1: Tabulka dominátorů jednotlivých uzlů (identifikovaných číslem post-order)

Pro výpočet dominance se budeme řídit článkem [2]. V článku jsou uvedeny dva způsoby výpočtu dominátorů. Oba způsoby jsou principiálně podobné, liší se však v náročnosti na výpočetní čas a paměťový prostor.

První ze způsobů vypočítá kompletní množinu dominance pro daný uzel. Toto řešení, ačkoliv je snáze implementovatelné, je však pro potřeby této práce nevyhovující jak pro výpočetní, tak i paměťovou složitost. Vzhledem k možnosti větších vstupů se proto budeme zabývat druhým způsobem.

Druhý způsob se opírá především o okamžité dominátory. Ty můžeme definovat jako uzel nejbližší uzlu p, který dominuje nad uzlem q a kde p != q. V našem případě je zde jedna výjimka, kterou tvoří počátek grafu, který je nastavíme jako okamžitý dominátor sám sebe. Okamžité

dominátory nám umožní definovat celou množinu dominance bez nutnosti mít ji u každého uzlu uloženou díky vlastnosti popsané v článku [2] *„dom(p) = {p} ∪ idom(p) ∪ idom(idom(p)) ... {0}*"

Zpočátku algoritmu je okamžitý dominátor všech uzlů nastaven na NULL. Nastavíme okamžitý dominátor počátečního uzlu na uzel samotný. Dále vytvoříme pomocnou proměnou, která nám bude značit jestli byla provedena změna. Poté budeme provádět cyklus dokud nenastane průchod beze změny. Pro každý uzel p v grafu, kromě uzlu start, najdeme první uzel q, jehož hrana vede do uzlu p. Následně budeme pomocí uzlu q provádět průnik s ostatními zdrojovými uzly. Pokud je výsledek průniku odlišný od současného okamžitého dominátoru uzlu p, nastavíme jej jako okamžitý dominátor a označíme do pomocné proměnné, že byla provedena změna.

Průnik uzlů je v tomto případě realizován ve formě dvou ukazatelů, které budeme postupně posouvat po jejich okamžitých dominátorech až do chvíle, kdy se číslo post-order ukazatelů bude rovnat. V takovém případě je výsledkem průniku kterýkoliv z ukazatelů. Pokud jeden z ukazatelů není nastaven (tzn. Pro daný uzel nebyl vypočten okamžitý dominátor) je výsledkem druhý z nich.

Výsledek výpočtu dominance na grafu z obrázku 7.1 je zachycen v tabulce 7.2.

Uzel p	0	1	2	3	4	5	6	7	8	9	10	11
idom(p)	0	0	2	2	1	4	5	6	7	8	9	1

Tabulka 7.2: Tabulka okamžitých dominátorů jednotlivých uzlů

7.3 Množina follow

Nyní již můžeme přistoupit k samotné tvorbě množiny follow. Ta vyjadřuje uzly, jež následují za uzlem p na stejné strukturální úrovni zanoření. Pro následný výpočet množiny follow využijeme vlastností množiny a to, jednak že a také velmi důležité vlastnosti, že jednotlivý uzel může být obsažen jen a pouze v jedné množině follow[1]. Tato vlastnost nám umožní nespecifikovat celou množinu pro každý uzel. Místo ní pro každý uzel p uložíme pouze odkaz na uzel q, v jehož množině follow se uzel p nachází.

Pro výpočet množiny follow začneme lineárně procházet redukovatelnou verzi grafu. Pro každý uzel p, kde p není typu start začneme procházet graf zpětně, čímž zaručíme, že pro každý uzel q bude platit že post-order(q) < post-order(p). Prvně ověříme, že head(q) = head(p). Pokud

[1] Množiny follow všech uzlů v grafu jsou párově disjunktní

tomu tak není a q není počáteční uzel v grafu, pak skočíme na uzel předcházející q.V opačném případě není uzel p v žádné množině follow. Poté se postup odlišuje dle typu instrukce q:

- **SLC, Start a Stop:** Pro tyto instrukce platí, že p je ve follow q když idom(p) = q.
- **Repeat:** Pro tento typ instrukcí platí, že p je ve follow q pokud head(idom(p)) = q.
- **Conditional:** Pro tyto instrukce platí, že p je ve follow q když idom(p) = q a zároveň do p vedou alespoň dvě dopředné hrany.

Výsledek výpočtu množin follow na grafu z obrázku 7.1 přehledně zachycuje tabulka 7.3. Z ní je patrné, že některé uzly nejsou v žádné množině follow. Toto řešení je správné, jelikož uzly, jež neodpovídají žádné množině jsou počáteční uzly bloku příkazů.

Uzel p	Follow(p)
0	1
1	4, 11
2	NULL
3	NULL
4	NULL
5	6
6	NULL
7	8
8	NULL
9	10
10	NULL
11	NULL

Tabulka 7.3: Tabulka množin follow

8 Tvorba výsledného grafu

Následující odstavce popisují způsoby tvorby výsledného strukturovaného grafu, který představuje jednu z možností výstupu programu.

8.1 Popis výsledného grafu

Až doposud jsme pracovali s jedním typem grafu. Ten je však pro správné zachycení struktury nevyhovující. Proto definujeme v této kapitole nový graf, který bude splňovat požadavky, kladené na výsledný program, a to:

1. Výsledný graf bude obsahovat více typů instrukcí než tomu bylo u grafu vstupního. Jednotlivé typy instrukcí jsou podrobně rozebrány v podkapitole 8.2.
2. U výsledného grafu musí jít snadno rozlišit úroveň zanoření instrukce.
3. Výsledný graf musí být lehce převoditelný do tvaru zdrojového kódu.

8.2 Typy instrukcí výsledného grafu

Zde jsou podrobně rozebrány jednotlivé instrukce výsledného grafu.

- **Typ Start**: Jedná se o počáteční uzel v grafu. Tato instrukce je v grafu pouze jediná. Nevedou do ní žádné hrany a jako jediná nemá nikdy přiřazený žádný label. Z instrukce start vede pouze jedna cesta a to na uzel na stejné úrovni zanoření.
- **Typ Stop:** Jedná se o koncový uzel grafu. Na rozdíl od uzlu start jich může být v grafu vícero. Uzel stop nemá nijak omezen počet vstupních hran, ale neobsahuje žádnou hranu vedoucí na následníka. Uzel stop představuje konec vykonávání programu bez ohledu na stupeň zanoření.
- **Typ SLC:** Jedná se o běžnou instrukcí grafu. Nemá omezen počet vstupních hran, ale má pouze jedinou výstupní. Ta je buď vedena na příkaz na stejné úrovni zanoření a nebo je prázdná.
- **Typ Repeat:** Jedná se o instrukci představující smyčku. Uzel repeat má dvě hrany, které z něj vedou. První z nich je hrana na jeho následníka, kterým se v tomto případě rozumí první uzel jdoucí za tělem smyčky se stejnou úrovní zanoření. Pokud žádny takový uzel není, je hrana prázdná. Druhá hrana vede na první uzel v těle smyčky. Uzly, které jsou součástí těla smyčky, musí být zařazeny v nižší úrovni zanoření.
- **Typ Conditional:** Jedná se o uzel grafu představující podmínku. Uzel má tři hrany, které z něj vedou. První dvě jsou hrany vedoucí na první příkazy v true a false klauzulích, které tvoří samostatné bloky na nižší úrovni zanoření.
- **Typ Break:** Jedná se o uzel grafu, představující přerušení cyklu. Tento uzel musí být

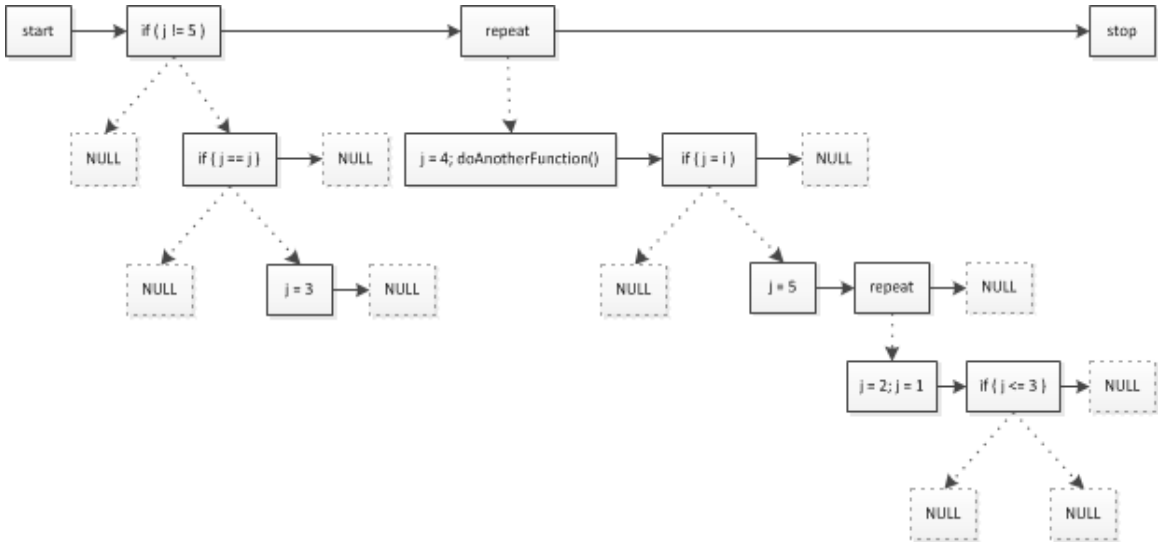
obsažen v rámci těla smyčky. Uzel break má jako parametr udané pořadí přerušujícího cyklu. Z tohoto uzlu nevede žádná hrana.

- **Typ Continue:** Jedná se o uzel grafu, představující skok na počátek těla smyčky. Tento uzel musí být součástí těla smyčky. Podobně jako uzel break, má i continue parametr, který určuje cyklus, na jehož začátek se z continue přechází.
- **Typ Goto:** Jedná se o uzel grafu představující skok. Uzel goto má pouze jedinou hranu, jež z něj vede. Tato hrana může vést na jakýkoliv uzel (kromě uzlu start) v grafu bez ohledu na úroveň zanoření. Tento uzel je dodán do grafu jen a pouze v případě, kdy nelze použit jiný typ uzlu.

8.3 Tvorba základní struktury

Prvním krokem k vytvoření výsledného strukturovaného grafu je vytvoření jeho kostry. Kostrou se rozumí graf s absencí větvících uzlů[1]. Pro jeho tvorbu využijeme mírně upravený algoritmus „GetForm“ z článku [1]. Pro potřeby doplnění větvících instrukcí v podkapitole 8.4 zároveň vytvoříme mapovaní uzlů finálního grafu na původní graf toku a naopak.

Při zpracovávání budeme procházet graf toku po jeho hranách s tím, že pro každý uzel p budeme zpracovávat prvně jeho následníky a pote sekvenčně zpracujeme uzly v jeho množině follow. Následníka v tomto případě mohou mít pouze uzly typu conditional a repeat. V případě těchto uzlů musíme ještě ověřit, že v jejich následníci nejsou v žádné množině follow. Pokud v nějaké jsou, nebudeme je zpracovávat a daný následník uzlu p zůstane prázdný.



Obrázek 8.1: Základní struktura grafu z obrázku 7.1

[1] Větvící instrukce jsou dle článku [1] instrukce break, continue a goto

8.4 Množina Reach

Pro správné doplnění větvících instrukcí je zapotřebí zjistit, kam každý uzel ve výsledném grafu může vést a to včetně následníků v jeho podstromě. To nám pomůže zjistit množina reach. Ta nám pro daný uzel p definuje uzly, na které je možno se z uzlu p dostat, ale které nejsou v rámci podstromu p.

Z definice výše vyplývá, že prvním úkolem tohoto kroku bude rozlišit jednotlivé podstromy ve výsledném grafu. Abychom tohoto dosáhli očíslujeme každý uzel p výsledného grafu a zároveň i nastavíme dosah p, nebo-li číslo posledního uzlu v podstromě uzlu p.

Z počátku nastavíme číslo již zpracovaných uzlů n na 0. Začneme zpracovávat jednotlivé uzly od počátku grafu. Pro každý uzel p nastavíme jeho číslo na n a inkrementujeme n o 1. Poté pokud má uzel p nějaké následníky[1] zpracujeme nejdříve je. V této fázi je n číslem posledního uzlu v podstromě p +1. Nastavíme tedy dosah p jako n – 1, načež pokud vede z uzlu p hrana na uzel stejného zanoření, pokračujeme ve zpracovávání s ní.

Nyní, když máme definovány podstromy jednotlivých uzlů, můžeme začít s výpočtem množiny reach. Pro každý uzel p z výsledného grafu se podíváme kam vedou jeho hrany v původním grafu toku. Pokud daná hrana vede na uzel q, jehož reprezentace není ve výsledném grafu v rámci podstromu p, přidáme q do množiny reach uzlu p. Pokud má uzel p následníky, zpracujeme přednostně je. Výsledná množina reach uzlu p se pak sestává ze sjednocení množin následníků s jeho vypočítaným dosahem, kde všechny uzly v množině reach(p) musí být mimo podstrom p. Číslování výsledného grafu a výpočet množiny reach na grafu z obrázku 8.1 přehledně zachycuje tabulka 8.1.

[1] Následníci jsou v tomto případě jednotlivé větve uzlů typu conditional a repeat

Uzel p	PostOrder(p)	Rozsah	Reach(p)
start	0	0 - 0	1
if (j != 5)	1	1 – 3	4, 6, 11
if (j == j)	2	2 – 3	6, 11
j = 3	3	3 – 3	11
repeat	4	4 – 10	11
j = 4; doAnotherFunction()	5	5 – 5	6
if (j == i)	6	6 – 10	4, 11
j = 5	7	7 – 7	8
repeat	8	8 – 10	4
j = 2; j = 1	9	9 – 9	10
if (j <= 3)	10	10 – 10	4, 8
stop	11	11 – 11	NULL

Tabulka 8.1: Tabulka rozsahu uzlů a jejich množiny reach

8.5 Doplnění větvících instrukcí

Posledním krokem, který nám zbývá provést je doplnění výsledného grafu o větvící instrukce. Zde musíme dbát na to, abychom do grafu nedodávali redundantní instrukce. V článku [1] je redundantní instrukce definována jako: „Větvící instrukce je redundantní v případě, kdy předává kontrolu instrukci, které by bylo dosaženo kdyby byla instrukce smazána.“

Abychom snadno zjistili, jestli daná instrukce bude redundantní či nikoliv, definujeme si pro každý uzel p z výsledného grafu okamžitého následníka q. Okamžitý následník je uzel, na který bude směřovat p, nebude-li použita žádná větvící instrukce. Ten zjistíme jednoduchým postupem. Pro každý uzel p ve výsledném grafu platí že, pokud má p následníka q na stejné úrovní zanoření, je q jeho okamžitým následníkem. Pokud nemá a je p součástí alespoň jednoho podstromu, pak jeho okamžitý následník je odvislý od typu prvního uzlu nejvnitřnějšího podstromu. Pokud je daný uzel typu repeat, je okamžitý následník uzlu p sám repeat. V opačném případě je použit okamžitý následník prvního uzlu podstromu.

Nyní již můžeme přistoupit k samotnému doplnění větvících instrukcí. Pro tento proces využijeme mírně upravené algoritmy „AddBranch“, „FixControl“ a „ChooseBranch“ z článku [1].

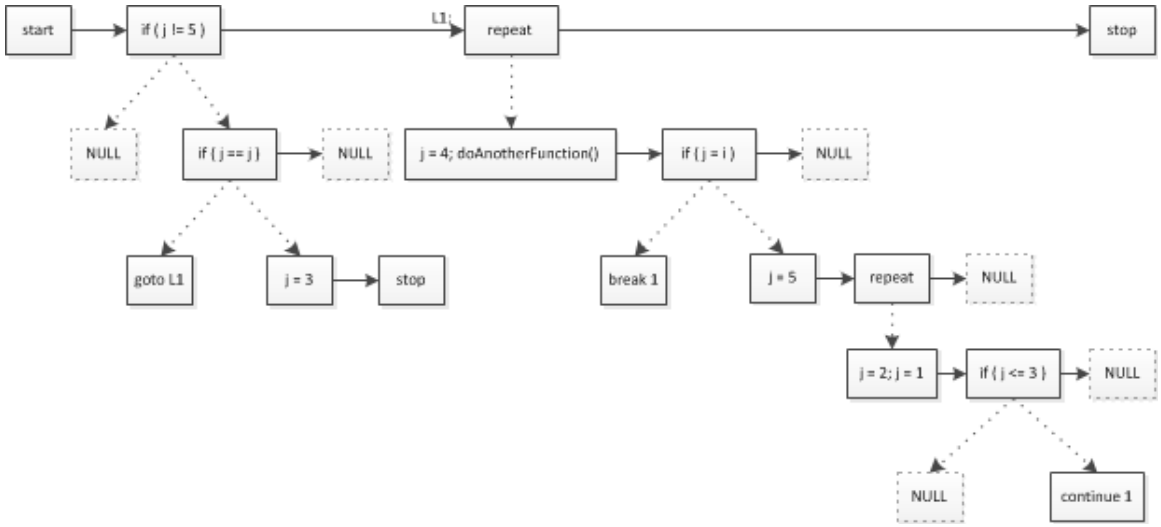
V první metodě „AddBranch“ spočítáme na výsledném grafu množinu reach, popsanou v podkapitole 8.4, načež ověříme, jestli výsledný graf má alespoň jeden nevětvící uzel. Pokud tento požadavek splní, zavolá se na první uzel v grafu metoda „FixControl“. V opačném případě

algoritmus skončí.

Druhá metoda „FixControl“ je vždy volána na uzel p. Pokud reach(p) obsahuje pouze jeden uzel q pak pomocí metody „ChooseBranch“ dodáme větvící uzel, pokud na q již nevede žádná hrana a pokud se okamžitý následník p nerovná q . Dále pokud je p typu repeat, začneme stejným způsobem zpracovávat jeho následníka. Pokud je zpracovávaný uzel p typu conditional, pak pro každou z jeho větví, která je prázdná[1] najdeme v původním grafu toku odpovídající uzel q ve výsledném grafu, na něhož má větev směřovat. Pokud q není okamžitým následníkem p, pak pomocí metody „ChooseBranch“ dodáme do větve vhodný uzel. U větví jež nejsou prázdné, začneme stejným způsobem zpracovávat uzel q, na který vedou. Pokud má uzel p následníka se stejnou úrovní zanoření, pak jej začneme tímto způsobem zpracovávát.

Poslední metoda „ChooseBranch“ je volána na uzel p a cíl uzlu q. Tato metoda vždy vytvoří větvící uzel, proto je důležité odfiltrovat nezbytnost vytvářeného uzlu v metodě předchozí. Úkolem této metody je zvolit, který z větvích uzlů vytvořit. Prvně do čísla n uložíme počet cyklů v nichž je uzel p zapozdřený. Pote ověříme každý cyklus od nejvnitřnějšího a pokud je jeho následník roven q, pak vytvoříme uzel typu break s parametrem n - (pořadí cyklu od nejvnitřnějšího – 1). Pokud q není rovno následníku cyklu, nýbrž jemu samému, pak vytvoříme uzel continue s týž parametrem, jako v předchozím případě. Pokud je q typu stop, pak vytvoříme pro danou větev typ uzlu stop. Pokud žádná z těchto variant nebude vyhovovat, pak zbývá jedině vytvořit uzel typu goto. V takovém případě ověříme, jestli má q nastavený label (tzn. Již na něj vede jiný skok z programu), pokud ne přiřadíme mu label, který zároveň nastavíme pro vytvářený uzel goto.

V této fázi již máme vytvořený výsledný graf, který odpovídá původnímu kódu. Tento graf je také jedna z možných forem výstupu z programu.



Obrázek 8.2: Finální graf z obrázku 8.1 doplněný o vetvící instrukce

8.6 Generace kódu

V případě, že z programu nechceme výstup ve formě grafu, ale ve formě kódu, je zapotřebí tento kód správně z grafu vygenerovat. Tento postup je poměrně přímočarý, kdy procházíme finální graf po jednotlivých uzlech s tím, že pro každý uzel vypíšeme odpovídající instrukci s odsazením dle stupně zanoření.

Jak je patrné z obrázku 8.2, může nastat případ kdy jedna z větví uzlu typu conditional nemá přiřazený žádný uzel pro jednu z větví. V takovém případě pokud je prázdná větev pravdivá klauzule, pak vynecháme klauzuli else. V opačném případě vypíšeme před obsah podmínky operátor negace a větve přehodíme, čímž opět můžeme vynechat klauzuli else.

Příklad vygenerovaného kódu 8.1:

```
start;
if (!(j!=5))
{
        if (j==j)
        {
                goto L1;
        }
        else
        {
                j=3;
                stop;
        }
}
repeat
{
        j=4;
        doAnotherFunction();
        L1: if (j==i)
        {
                break 1;
        }
        else
        {
                j=5;
                repeat
                {
                        j=2;
                        j=1;
                        if (!(j<=3))
                        {
                                continue 1;
                        }
                }
        }
}
stop;
```

9 Finální program

V následujících odstavcích bude rozebrán výsledný program a jeho způsob použití.

9.1 Popis Programu

Program je koncipován objektově jako pomocná knihovna, kdy je každý objekt implementován ve svém samostatném souboru, což umožní jednoduší změnu částí knihovny.

Jelikož je zde předpoklad, že graf bude generován z regulérního programovacího jazyka a nikoliv z vstupního jazyka programu, je část lexikální a syntaktické analýzy řešena samostatně.

Pro vývoj programu byl použit zdarma dostupný program Code::Blocks.

9.2 Použití programu

Program je v současném provedení možno použít dvěma způsoby. První způsob je regulerní spuštění kompilovaného programu, přiloženého jako ukázka použiti na CD.

Ten je možno spustit bez parametru, kdy se program pokusí převést obsah souboru “in.txt” do jeho strukturované podoby. Obsah souboru “in.txt” musí splňovat jazykové požadavky, definované v kapitolách 3 a 4. Program je také možno spustit s parametrem, který představuje cestu ke vstupnímu souboru.

Při použití programu jako knihovny je nutné do programu zařadit veškeré zodpovědné objekty. Toho docílíme zahrnutním knihovny mainlib:

```
#include “mainlib.hpp”
```

Pokud pro program definujeme vlastni lexikalní a syntaktický analyzátor, je zapotřebí před samotným zahrnutím knihovny zadat:

```
#define LOADER_HPP
```

Přiklad použití knihovny s využitím lexikálního a syntaktického analyzátoru knihovny je ukázáno na příkladu 9.1

Příklad použití knihovny pro generaci vstupu do konzole 9.1:

```
LexicalAnalyser* lex = new LexicalAnalyser(filename, INPUT_TYPE_FILE);
if (lex->Analyse())
{
      SyntacticAnalyser* syn = new SyntacticAnalyser(lex->GetTokens());
      if (syn->Analyse())
      {
            FlowGraph *flowgraph = new FlowGraph();
            flowgraph->SetGraph(syn->GetGraph());
            flowgraph->Process();
            flowgraph->GenerateCode(flowgraph->GetResultStart());
      }
}
```

V případě, že nechceme generovat výstup do konzole, ale chceme k dispozici výsledný graf, je prvně nutné nahrát třídu `vector` ze standartní knihovny jazyka C. Poté již v kódu stačí nahradít funkci

```
flowgraph->GenerateCode( flowgraph->GetResultStart() );
```

za funkci jež vrací vector odkazů na jednotlivé uzly výsledného grafu, s tím že odkaz na pozici 0 bude vždy ukazovat na počáteční uzel typu start.

```
Vector< ResultNode* > result_graph = flowgraph->GetResultGraph();
```

10 Závěr

Cíl této práce, kterým bylo vytvoření programu pro automatické strukturování nestrukturovaného kódu, byl splněn. V programu je stále mnoho věcí, které nabízejí potencionální prostor pro zlepšení, kupříkladu podpora blokových příkazů pro lexikální a syntaktický analyzátor či hlubší optimalizace výsledného kódu, ovšem i na současné verzi lze vidět správnost a efektivitu řešení.

Tato práce, ačkoliv její vypracování se ukázalo jako poměrně náročné, obohatila mé dosavadní programátorské zkušenosti svým extenzivním zpracováváním grafů toku, což hodnotím jako velmi přínosné a také podstatně ovlivnila můj přístup k obecné problematice grafů toku o kterých nyní uvažuji v rámci témat dalšího studia.

11 Literatura

[1] – BAKER, Brenda S. An Algorithm for Structuring Flowgraph. *Journal of Association for Computing Machinery,* January 1977, Vol. 24*,* no. 1, s. 98-120

[2] – COOPER, Keith D. - HARVEY, Timothy J - KENNEDY, Ken A simple, fast dominance algorithm. Technical report (Rice Computer Science TR-06-33870).
URL: <http://www.cs.rice.edu/~keith/EMBED/dom.pdf> [cit. 2013-02-12]

[3] – Dalbey, John Structured Programming
URL: <http://users.csc.calpoly.edu/~jdalbey/308/Resources/StructuredProgramming.pdf>
[cit. 2013-04-29]

A. Příloha

Vzorové vstupy

A.1. Vstup 1

Název souboru se vstupem na cd: „in1.txt“

Vstup pro program A.1:

```
start;
if ( i != 10 ) goto L2;
j = 1;
L2: goto L3;
someFunc();
L3: j + 3;
Function();
Function2();
Function3();
if ( a == 5 ) goto L3;
stop;
```

Vygenerovaný výstup pro vstup A.1:

```
start;
if (!(i!=10))
{
        j=1;
}
repeat
{
        j+3;
        Function();
        Function2();
        Function3();
        if (!(a==5))
        {
                break 1;
        }
}
stop;
```

A.2. Vstup 2

Název souboru se vstupem na cd: „in2.txt“

Vstup pro program A.2:

```
i = 0;
if ( i > 10 ) goto L1;
L2: MakeSmth();
L4:ForgeSmth();
if ( SmthWorks() ) goto L4;
i + 1;
if ( 1 == 1 ) goto L7;
i = 6;
goto L2;
L1: MakeCalculation();
L3: if ( c != i ) goto L1;
i - 15;
if ( c >= i ) goto L2;
goto L6;
L6: goto L7;
L7: goto L5;
L5:stop;
```

Vygenerovaný výstup pro vstup A.2:

```
start;
i=0;
if (i>10)
{
        repeat
        {
                MakeCalculation();
                if (!(c!=i))
                {
                        break 1;
                }
        }
        i-15;
        if (!(c>=i))
        {
                stop;
        }
}
repeat
{
        MakeSmth();
        repeat
        {
                ForgeSmth();
                if (!(SmthWorks))
                {
                        break 2;
                }
        }
        i+1;
        if (1==1)
        {
                break 1;
        }
        else
        {
                i=6;
        }
}
stop;
```

A.3. Vstup 3

Název souboru se vstupem na cd: „in3.txt“

Vstup pro program A.3:

```
start;
L4: j = 1;
L1: goto L2;
j = 2;
L2: goto L3;
Functionx();
L3: if ( a = 1 ) goto L4;
i = j;
ii = j;
j = 6;
L5: j * 5;
if ( someFunc() ) goto L6;
goto L7;
L6:j=5;
if ( checkFunc() ) goto L9;
goto L7;
L7: if ( p == 0 ) goto L8;
goto L5;
L8: j = 666;
Function1();
if ( someFunc() ) goto L5;
j=5;
j=2;
j = 0;
Function2();
if ( i == p ) goto L8;
j=2;
L9:stop;
```

Vygenerovaný výstup pro vstup A.3:

```
start;
repeat
{
        j=1;
        if (!(a=1))
        {
                break 1;
        }
}
i=j;
ii=j;
j=6;
repeat
{
        j*5;
        if (someFunc)
        {
                j=5;
                if (checkFunc)
                {
                        break 1;
                }
        }
        if (p==0)
        {
                repeat
                {
                        j=666;
                        Function1();
                        if (someFunc)
                        {
                                continue 1;
                        }
                        else
                        {
                                j=5;
                                j=2;
                                j=0;
                                Function2();
                                if (!(i==p))
                                {
                                        j=2;
                                        stop;
                                }
                        }
                }
        }
}
stop;
```

B. Příloha

Obsah CD

B.1. Adresářová Struktura

- **src –** Tento adresář obsahuje zdrojové kódy práce
- **samples –** Zde se nachází vzorové příklady k práci.